by
Ray Duncan

Power Programing

Creating a DPMI-Based DOS Extender Of Your Own

■ **DOS extenders are technologically fascinating and, luckily, the *Windows* 3.0. implementation of the DOS Protected Mode Interface gives us a low-cost test-bed for DOS extender experimentation.**

Life isn't easy for the authors of commercial DOS extenders these days. Their products must be capable of running on a completely standalone basis, taking full control of the 286, 386, and 486 processors in protected mode, and maintaining all the special CPU control registers, descriptor tables, and page tables. And their brainchildren must also be able to coexist harmoniously with memory managers or device drivers that implement any of the existing protocols or standards for extended memory allocation—be it top-down (INT 15h), bottom-up (VDISK), XMS (HIMEM.SYS), VCPI (*QEMM-386* or *386MAX*), or DPMI (*Microsoft Windows* 3.0). Developers of DOS extenders must also be masters of the various PC hardware architectures, with all their peculiarities and ramifications for mode switching, toggling of the A20 address line, and the like. This certainly isn't a job description I'd envy, even if it does carry its own unique rewards (the staff of Phar Lap Software has been growing, I suspect, as quickly as the Microsoft OS/2 development staff has been shrinking).

But DOS extenders are a technologically fascinating topic, and fortunately events have conspired to place a low-cost test-bed for DOS extender experimentation at our disposal. I refer, of course, to the implementation of the DOS Protected Mode Interface (DPMI) in *Windows* 3.0. First, let's review how "real" DOS extenders work, and then we'll implement a simple little DPMI-based DOS extender of our own.

THE TWO FACES OF A DOS EXTENDER

A typical DOS extender that you might buy from Phar Lap or Rational Systems can be thought of as having two main sections: an initialization component and an interrupt handler component. The initialization portion of a DOS extender gains control of the machine when the DOS extender is loaded by DOS in real mode as a result of a user command or an INT 21h, function 4Bh (EXEC) function call by another program. In most cases, the DOS extender and the protected-mode application (such as *Mathematica, AutoCAD,* or *Lotus 1-2-3,* Release 3.0) are bound together in the same .EXE file. However, the only part of the file visible to DOS is the DOS extender, because it's the only part described by the .EXE file header and relocation table.

The initialization routine has a lot of work to do before the protected-mode application can start running. It must probe the environment and determine whether it's running on the bare hardware (aside from DOS) or in the presence of one of the several species of memory managers. It must allocate some extended memory and load the application's code and data, performing any necessary relocations and fixups. It must allocate additional conventional memory (memory below 640K) for communication with MS-DOS and the ROM BIOS. It must build the necessary global and local descriptor tables to support protected-mode addressing, which includes mapping some descriptors onto "magic" memory areas like the PSP, the environment block, the video refresh buffer, and the ROM BIOS data area, for the convenience of the application. It must switch the CPU from real mode into protected mode. And it must take full command of the CPU's interrupt subsystem by building an interrupt descriptor table, reprogramming the 8259 interrupt controllers, and installing handlers for every type of interrupt that might occur in protected mode.

From the DOS extender's point of view, there are three types of interrupts to worry about. The first group consists of the CPU faults or exceptions, usually caused by a program error but sometimes by a hardware error (for example, a nonmaskable interrupt due to a RAM parity error) or an unexpected result on the numeric coprocessor. The second group consists of the external hardware interrupts—signals that a peripheral device has received data, is ready to accept more data, or has encountered some other condition that needs attention. The third group are the software interrupts, which occur when an application program executes an INT instruction. Control over software interrupts is a particularly crucial aspect of DOS extender initialization, because it allows the DOS extender to function transparently by intercepting the application's requests for ROM BIOS and MS-DOS services.

The exact order in which the DOS extender will perform these initialization chores depends in part on the philosophy of the implementer and in part on the

environment in which the DOS extender will run. For example, XMS and VCPI memory managers require that the DOS extender make its memory allocation calls in real mode, while requests to a DPMI host for memory must be made in protected mode. Similarly, techniques for mode switching differ with the PC architecture (PC/AT, PS/2, and so forth) and with the presence or absence of XMS, VCPI, and DPMI. The only things we can say for sure about the initialization sequence are that a minimal global descriptor table (GDT) must be constructed in real mode and that the CPU must be in protected mode before control passes to the entry point of the actual application program.

Once the application is launched, the DOS extender's initialization section is dead code; the memory it occupies can be reclaimed and reused. The application's code now becomes the center of the action. The remaining parts of the DOS extender are activated only upon the occurrence of one of the three kinds of interrupts listed earlier. In this respect, the DOS extender is much like a real-mode terminate-and-stay-resident (TSR) program. But the DOS extender's status after initialization is somewhat more slippery than that of your average TSR.

From MS-DOS's perspective, the DOS extender is the only program that's running, and it's a perfectly ordinary program at that: it lives in the memory that MS-DOS allocated for it, and it makes normal MS-DOS function calls. From the protected-mode application's point of view, the DOS extender is nearly invisible (in fact, the more invisible, the better the DOS extender); the application requests MS-DOS and ROM BIOS services by executing the usual software interrupts, and somehow the right things happen—even though the interrupts are being executed in protected mode and the application is passing virtual addresses of data and buffers lying in extended memory.

The fully initialized DOS extender can be thought of simply as a grab bag of interrupt service routines.

Looking at the DOS extender from the inside out (the viewpoint of its author), the fully initialized DOS extender can be thought of rather simply as a grab bag of interrupt service routines.

When a CPU fault or exception occurs, the DOS extender usually has little recourse but to terminate the protected mode application, unless the application has explicitly registered its own handler for the interrupt. If the application has gone so far awry as to cause a stack underflow or general protection (GP) fault, for example, it's obviously ailing and will only cause more trouble if it's allowed to continue. (Interpreters and debuggers are obvious exceptions.)

External hardware interrupts are treated differently. If the protected-mode application hasn't registered a handler, the interrupt is ''reflected'' to the original real-mode owner of the interrupt. In other words, the DOS extender fields the inter-

TINYDOSX.ASM **1 of 4**

```
        title   TINYDOSX DPMI-Based Tiny DOS Extender
        page    55,132

; TINYDOSX.ASM  Tiny DPMI-Based DOS Extender
; Copyright (C) 1990 Ziff Davis Communications
; PC Magazine * Ray Duncan

stdin   equ     0                       ; standard input handle
stdout  equ     1                       ; standard output handle
stderr  equ     2                       ; standard error handle
cr      equ     0dh                     ; ASCII carriage return
lf      equ     0ah                     ; ASCII line feed

DGROUP  group   _DATA

_DATA   segment word public 'DATA'

modesw  dd      0                       ; far pointer to DPMI mode
                                        ;   switch entry point
int0dv  dd      0                       ; address of previous
                                        ;   GP fault handler
int21v  dd      0                       ; address of previous
                                        ;   Int 21H handler
realseg dw      0                       ; segment of real mode buffer
realsel dw      0                       ; selector for real mode buffer

gpfmsg  db      cr,lf,lf
        db      'TINYDOSX: general protection fault!'
        db      cr,lf
gpfmsg_len equ  $-gpfmsg

abmsg   db      cr,lf,lf
        db      'TINYDOSX: unsupported DOS function!'
        db      cr,lf
abmsg_len equ   $-abmsg

regs    label   word                    ; real mode register structure
                                        ; for DPMI translation services
regDI   label   word                    ; 00H DI or EDI
regEDI  dd      0
regSI   label   word                    ; 04H SI or ESI
regESI  dd      0
regBP   label   word                    ; 08H BP or EBP
regEBP  dd      0
        dd      0                       ; 0CH (reserved)
regBX   label   word                    ; 10H BX or EBX
regEBX  dd      0
regDX   label   word                    ; 14H DX or EDX
regEDX  dd      0
regCX   label   word                    ; 18H CX or ECX
regECX  dd      0
regAX   label   word                    ; 1CH AX or EAX
regEAX  dd      0
cpuFLAGS dw     0                       ; 20H cpu status flags
regES   dw      0                       ; 22H ES
regDS   dw      0                       ; 24H DS
regFS   dw      0                       ; 26H FS
regGS   dw      0                       ; 28H GS
regIP   dw      0                       ; 2AH IP (CS:IP ignored by
regCS   dw      0                       ; 2CH CS  DPMI function 0300H)
regSP   dw      0                       ; 2EH SP (SS:SP=0 to have DPMI
regSS   dw      0                       ; 30H SS  host supply a stack)

protDX  dw      0                       ; save protected mode DX
protSI  dw      0                       ; save protected mode SI
protES  dw      0                       ; save protected mode ES

dispatch label  word                    ; Int 21H dispatch table
        dw      offset _TEXT:fxn00h     ; fxn 00H terminate
        dw      offset _TEXT:fxn01h     ; fxn 01H char input+echo
        dw      offset _TEXT:fxn02h     ; fxn 02H char output
        dw      offset _TEXT:fxn03h     ; fxn 03H aux input
        dw      offset _TEXT:fxn04h     ; fxn 04H aux output
        dw      offset _TEXT:fxn05h     ; fxn 05H printer output
        dw      offset _TEXT:fxn06h     ; fxn 06H raw console I/O
        dw      offset _TEXT:fxn07h     ; fxn 07H raw input no echo
        dw      offset _TEXT:fxn08h     ; fxn 08H char input no echo
        dw      offset _TEXT:abort      ; fxn 09H
        dw      offset _TEXT:abort      ; fxn 0AH
        dw      offset _TEXT:fxn0bh     ; fxn 0BH input status
        dw      offset _TEXT:abort      ; fxn 0CH
        dw      offset _TEXT:fxn0dh     ; fxn 0DH disk reset
        dw      offset _TEXT:fxn0eh     ; fxn 0EH select disk
        dw      offset _TEXT:abort      ; fxn 0FH
        dw      offset _TEXT:abort      ; fxn 10H
        dw      offset _TEXT:abort      ; fxn 11H
        dw      offset _TEXT:abort      ; fxn 12H
        dw      offset _TEXT:abort      ; fxn 13H
        dw      offset _TEXT:abort      ; fxn 14H
        dw      offset _TEXT:abort      ; fxn 15H
        dw      offset _TEXT:abort      ; fxn 16H
        dw      offset _TEXT:abort      ; fxn 17H
        dw      offset _TEXT:abort      ; fxn 18H
        dw      offset _TEXT:fxn19h     ; fxn 19H get current drive
        dw      offset _TEXT:abort      ; fxn 1AH
        dw      offset _TEXT:fxn1bh     ; fxn 1BH get cur. drive data
        dw      offset _TEXT:fxn1ch     ; fxn 1CH get drive data
        dw      offset _TEXT:abort      ; fxn 1DH
        dw      offset _TEXT:abort      ; fxn 1EH
        dw      offset _TEXT:abort      ; fxn 1FH
        dw      offset _TEXT:abort      ; fxn 20H
        dw      offset _TEXT:abort      ; fxn 21H
        dw      offset _TEXT:abort      ; fxn 22H
        dw      offset _TEXT:abort      ; fxn 23H
        dw      offset _TEXT:abort      ; fxn 24H
        dw      offset _TEXT:abort      ; fxn 25H
        dw      offset _TEXT:abort      ; fxn 26H
        dw      offset _TEXT:abort      ; fxn 27H
        dw      offset _TEXT:abort      ; fxn 28H
        dw      offset _TEXT:abort      ; fxn 29H
        dw      offset _TEXT:fxn2ah     ; fxn 2AH get date
        dw      offset _TEXT:fxn2bh     ; fxn 2BH set date
        dw      offset _TEXT:fxn2ch     ; fxn 2CH get time
```

Figure 1: Here's the source code for a skeleton DPMI 0.9-based DOS extender that can be linked with small-model C programs.

rupt, saves the CPU state, switches the CPU into real mode, and reissues the interrupt with an INT instruction. When the real-mode handler executes an interrupt return (IRET), the DOS extender recovers control, switches the CPU back into protected mode, restores the CPU state, and then issues its own IRET so that the application can continue.

Which brings us to the last category of interrupt that the DOS extender must dispose of: interrupts that are explicit MS-DOS or ROM BIOS function requests by the application. There are a number of different software interrupts that the DOS extender must be prepared to cope with: MS-DOS's INT 20h through INT 2Fh, the ROM BIOS video driver INT 10h, serial port driver INT 14h, keyboard driver INT 16h, printer driver INT 17h, Microsoft Mouse driver INT 33h, and so on. Each one of these interrupts provides a pathway to a host of different subfunctions, typically selected by a value in register AH. For instance, more than 80 functions (both documented and undocumented) are defined for INT 21h. As I explained in the last column, these functions can be divided into four classes:

- functions that require little more than a mode switch before passing them down to MS-DOS or the ROM BIOS;
- functions that address application buffers and therefore require data movement and address translation before they can be reissued to MS-DOS or the ROM BIOS;
- functions that must be completely replaced to make them meaningful in protected mode; and
- function calls that are unique to the DOS extender itself and provide special services that have no equivalents in MS-DOS or the ROM BIOS.

> In any event, the DOS extender's handling of an MS-DOS or ROM BIOS service request is easy to visualize.

There is also an implicit fifth class of functions, which I didn't mention previously: the MS-DOS functions that the DOS extender author simply chooses not to support because they're either too dangerous or not worth the hassle. The FCB file functions, the direct disk I/O functions, and some of the undocumented DOS functions that vary tremendously from one version of DOS to another are good examples.

In any event, the DOS extender's handling of an MS-DOS or ROM BIOS service request is easy to visualize. An umbrella routine is entered first; it saves the CPU flags and all the general and segment registers for later reference. The umbrella routine then decodes the function request by using the function number in AH as an index into a jump table, and then passes control to a handler that is specific to the function type. Functions that pass all parameters in registers and don't reference data in extended memory can fall through to another umbrella handler that performs the necessary mode switching and reissues the interrupt in real mode. Nearly all functions that pass addresses of buffers or pass parameters by reference must be handled on an individual basis, because there's regrettably little symmetry of structures or register usage across DOS and ROM BIOS function calls. Of course, memory management

TINYDOSX.ASM — 2 of 4

```
        dw      offset _TEXT:fxn2dh     ; fxn 2DH set time
        dw      offset _TEXT:fxn2eh     ; fxn 2EH set verify flag
        dw      offset _TEXT:abort      ; fxn 2FH
        dw      offset _TEXT:fxn30h     ; fxn 30H get DOS version
        dw      offset _TEXT:abort      ; fxn 31H
        dw      offset _TEXT:abort      ; fxn 32H
        dw      offset _TEXT:fxn33h     ; fxn 33H get/set break flag
        dw      offset _TEXT:abort      ; fxn 34H
        dw      offset _TEXT:abort      ; fxn 35H
        dw      offset _TEXT:fxn36h     ; fxn 36H get drive info
        dw      offset _TEXT:abort      ; fxn 37H
        dw      offset _TEXT:error      ; fxn 38H
        dw      offset _TEXT:fxn39h     ; fxn 39H create directory
        dw      offset _TEXT:fxn3ah     ; fxn 3AH delete directory
        dw      offset _TEXT:fxn3bh     ; fxn 3BH select directory
        dw      offset _TEXT:fxn3ch     ; fxn 3CH create file
        dw      offset _TEXT:fxn3dh     ; fxn 3DH open file
        dw      offset _TEXT:fxn3eh     ; fxn 3EH close file
        dw      offset _TEXT:fxn3fh     ; fxn 3FH read file
        dw      offset _TEXT:fxn40h     ; fxn 40H write file
        dw      offset _TEXT:fxn41h     ; fxn 41H delete file
        dw      offset _TEXT:fxn42h     ; fxn 42H seek
        dw      offset _TEXT:fxn43h     ; fxn 43H get/set attributes
        dw      offset _TEXT:error      ; fxn 44H
        dw      offset _TEXT:fxn45h     ; fxn 45H dup handle
        dw      offset _TEXT:fxn46h     ; fxn 46H redirect handle
        dw      offset _TEXT:fxn47h     ; fxn 47H get cur. directory
        dw      offset _TEXT:error      ; fxn 48H
        dw      offset _TEXT:error      ; fxn 49H
        dw      offset _TEXT:error      ; fxn 4AH
        dw      offset _TEXT:error      ; fxn 4BH
        dw      offset _TEXT:fxn4ch     ; fxn 4CH terminate
        dw      offset _TEXT:abort      ; fxn 4DH
        dw      offset _TEXT:error      ; fxn 4EH
        dw      offset _TEXT:error      ; fxn 4FH
        dw      offset _TEXT:abort      ; fxn 50H
        dw      offset _TEXT:abort      ; fxn 51H
        dw      offset _TEXT:abort      ; fxn 52H
        dw      offset _TEXT:abort      ; fxn 53H
        dw      offset _TEXT:fxn54h     ; fxn 54H get verify flag
        dw      offset _TEXT:abort      ; fxn 55H
        dw      offset _TEXT:error      ; fxn 56H
        dw      offset _TEXT:fxn57h     ; fxn 57H get/set file date
        dw      offset _TEXT:error      ; fxn 58H
        dw      offset _TEXT:abort      ; fxn 59H
        dw      offset _TEXT:fxn5ah     ; fxn 5AH create temp file
        dw      offset _TEXT:fxn5bh     ; fxn 5BH create unique file
        dw      offset _TEXT:fxn5ch     ; fxn 5CH lock/unlock
        dw      offset _TEXT:abort      ; fxn 5DH
        dw      offset _TEXT:error      ; fxn 5EH
        dw      offset _TEXT:error      ; fxn 5FH
        dw      offset _TEXT:abort      ; fxn 60H
        dw      offset _TEXT:abort      ; fxn 61H
        dw      offset _TEXT:abort      ; fxn 62H
        dw      offset _TEXT:error      ; fxn 63H
        dw      offset _TEXT:abort      ; fxn 64H
        dw      offset _TEXT:error      ; fxn 65H
        dw      offset _TEXT:error      ; fxn 66H
        dw      offset _TEXT:error      ; fxn 67H
        dw      offset _TEXT:fxn68h     ; fxn 68H commit file
        dw      offset _TEXT:abort      ; fxn 69H
        dw      offset _TEXT:abort      ; fxn 6AH
        dw      offset _TEXT:abort      ; fxn 6BH
        dw      offset _TEXT:error      ; fxn 6CH
        dw      offset _TEXT:abort      ; fxn 6DH
        dw      offset _TEXT:abort      ; fxn 6EH
        dw      offset _TEXT:abort      ; fxn 6FH

_DATA   ends

_TEXT   segment byte public 'CODE'

        assume  cs:_TEXT,ds:DGROUP
;
; Initialization routine for the Tiny DOS Extender.  First we test for
; the presence of a DPMI host, get the address of the mode switch entry
; point, and request the switch to protected mode.  Then we install
; a handler for GP faults to circumvent the Win 3 brain-damaged dialog
; box, and allocate some memory below 1 MB to use as a buffer for
; communication with DOS.  Finally we install our own Int 21H handler
; so we can service DOS calls from the protected mode application.
;
        public  initdosx
initdosx proc   near

        mov     ax,1687h                ; get address of DPMI
        int     2fh                     ;   mode switch entry point
        or      ax,ax                   ; bail out if no DPMI
        jnz     init9
        mov     word ptr modesw,di      ; save far pointer to
        mov     word ptr modesw+2,es    ;   DPMI entry point

        mov     bx,si                   ; allocate DPMI private data
        mov     ah,48h                  ; area below 1 MB boundary
        int     21h
        jc      init9                   ; jump, allocation failed

        mov     es,ax                   ; pass segment of data area
        mov     ax,0                    ; bit 0=0 indicates 16-bit app
        call    modesw                  ; switch to protected mode

        mov     ax,0202h                ; get address of previous
        mov     bl,0dh                  ; owner of GP fault vector
        int     31h
        mov     word ptr int0dv,dx      ; save as far pointer
        mov     word ptr int0dv+2,cx

        mov     ax,0203h                ; install our GP fault handler
        mov     bl,0dh
        mov     cx,cs                   ; CX:DX = handler address
        mov     dx,offset _TEXT:gpfisr
        int     31h
        jc      init9                   ; jump, couldn't install

        mov     ax,0100h                ; allocate 64 KB buffer in
        mov     bx,1000h                ; conventional memory for
        int     31h                     ; communication with DOS
        jc      init9                   ; jump, allocation failed
        mov     realseg,ax              ; save segment of block
        mov     realsel,dx              ; save selector for block

        mov     ax,0204h                ; get address of previous
```

function calls must be trapped and serviced entirely within the DOS extender, and this can be a fair amount of work in itself (particularly if the DOS extender supports virtual memory).

One more function that bears mentioning is INT 21h, function 4Ch, which an application calls to terminate itself. When the DOS extender sees this function request, it must deallocate its (and the application's) extended memory, release the interrupt subsystem, and remove any other little tendrils it may have inserted elsewhere. This is tricky business, because the DOS extender—which has been playing the role of a little protected-mode operating system that uses DOS as a file system slave—must quietly and gracefully put everything back *exactly* as it was before the DOS extender was loaded, leaving the system completely stable. Finally, the DOS extender must switch the CPU into real mode for the last time and itself call INT 21h, function 4Ch, so that DOS knows to free all conventional memory, file handles, and other resources that were assigned to the DOS extender.

A SIMPLE DPMI-BASED DOS EXTENDER

I always find source code much more enlightening than windy explanations, so I've illustrated the foregoing discussion of DOS extenders with TINYDOSX.ASM (Figure 1), a skeleton DOS extender that you can use in your own programs. TINYDOSX relies only on the existence of a DPMI 0.9 host, such as the one found in *Windows* 3.0; it can be linked into any small-model C program and will cause that program to execute in protected mode—provided you don't call any runtime library functions that include self-modifying code, use segment registers for scratch storage, perform segment register arithmetic, or in general execute an instruction that will result in a general protection fault. Although *Windows* 3.0 includes a DOS extender of its own, TINYDOSX doesn't use it and will run just as well in other DPMI, Version 0.9, environments (at least theoretically; no other such environments exist for testing at this time).

Memory management function calls must be trapped and serviced entirely within the DOS extender, which can be a fair amount of work.

The initialization portion of TINYDOSX, embodied in the routine INITDOSX, is straightforward. To keep things simple, we allow the C application to get control first in real mode, and require it to explicitly call the DOS extender, rather than the other way around. INITDOSX first calls INT 2Fh, function 1687h to find out whether a DPMI host is present, and if so, the address of the mode switch entry point. If a DPMI host is not found, INITDOSX bails out with an error message; otherwise, it proceeds to allocate the private data area required by the DPMI host and then requests the switch into protected mode.

Once it's running in protected-mode, INITDOSX installs protected mode handlers for MS-DOS INT 21h and for general protection faults (so that *Windows* won

TINYDOSX.ASM

```
        mov     bl,21h                  ; owner of Int 21H vector
        int     31h
        mov     word ptr int21v,dx      ; save as far pointer
        mov     word ptr int21v+2,cx

        mov     ax,0205h                ; install our Int 21H handler
        mov     bl,21h
        mov     cx,cs                   ; CX:DX = handler address
        mov     dx,offset _TEXT:doscall
        int     31h

        xor     ax,ax                   ; return in protected mode
        ret                             ; AX = 0 to signal success

init9:  mov     ax,-1                   ; return with AX <> 0 to
        ret                             ; signal initialization failure

initdosx endp
;
; Interrupt service routine for GP faults. Entered by a far call
; from DPMI host with CS:IP, flags, CPU error code on stack.
; We force transfer to our error message routine by changing
; return address in the stack frame.
;
gpfisr  proc    far

        push    bp                      ; point CS:IP in stack frame to
        mov     bp,sp                   ; GP fault error message routine
        mov     word ptr [bp+8],offset _TEXT:gpferr
        pop     bp
        ret

gpfisr  endp
;
; This routine gains control after the GPFISR returns to the DPMI host.
; It simply displays an error message and terminates cleanly, subverting
; Win 3's "Application has violated system integrity" dialog box.
;
gpferr  proc    near

        mov     dx,offset DGROUP:gpfmsg ; display GP fault message
        mov     cx,gpfmsg_len           ; on standard output
        mov     bx,stdout
        mov     ah,40h
        int     21h
        mov     ax,4c01h                ; and terminate with
        int     21h                     ; nonzero return code

gpferr  endp
;
; The DOSCALL routine is the runtime portion of the Tiny DOS Extender.
; It traps Int 21H requests in protected mode and performs any necessary
; mode switching, data movement, and address translation on a
; function-by-function basis.  Anything DOSCALL doesn't want to handle,
; it either fails by setting the Carry flag and returning, or
; it aborts the current program.  In particular, all FCB-related
; functions are aborted.  When a termination function is detected,
; the interrupt handlers are unhooked and the function call is
; passed down to the DPMI host so that all other protected mode
; resources will be deallocated.
;
```

```
doscall proc    far

        push    bx                      ; save register BX
        mov     bl,ah                   ; function number * 2
        xor     bh,bh
        cmp     bx,6fh                  ; function no. too big?
        ja      abort                   ; yes, bail out
        add     bx,bx                   ; no, branch through table
        jmp     [dispatch+bx]           ; to function handler

abort:                                  ; unsupported DOS function
        mov     dx,offset DGROUP:abmsg  ; display error message
        mov     cx,abmsg_len
        mov     bx,stdout
        mov     ah,40h
        int     21h
        mov     ax,4c01h                ; and exit to DOS
        int     21h

error:                                  ; unsupported DOS function
        push    bp                      ; set Carry flag in stack
        mov     bp,sp                   ;   frame to indicate
        or      word ptr [bp+6],1       ;   function failed
        pop     bp                      ; load AX = error code for
        mov     ax,1                    ;   "invalid function number"
        iret                            ; return to application

                                        ; common handling for entirely
                                        ;   register-based functions
fxn01h:                                 ; function 01H: char input+echo
fxn02h:                                 ; function 02H: char output
fxn03h:                                 ; function 03H: aux input
fxn04h:                                 ; function 04H: aux output
fxn05h:                                 ; function 05H: printer output
fxn06h:                                 ; function 06H: raw console I/O
fxn07h:                                 ; function 07H: raw input no echo
fxn08h:                                 ; function 08H: char input no echo
fxn0bh:                                 ; function 0BH: input status
fxn0dh:                                 ; function 0DH: disk reset
fxn0eh:                                 ; function 0EH: select disk
fxn19h:                                 ; function 19H: get current drive
fxn1bh:                                 ; function 1BH: get cur. drive data
fxn1ch:                                 ; function 1CH: get drive data
fxn2ah:                                 ; function 2AH: get date
fxn2bh:                                 ; function 2BH: set date
fxn2ch:                                 ; function 2CH: get time
fxn2dh:                                 ; function 2DH: set time
fxn2eh:                                 ; function 2EH: set verify flag
fxn30h:                                 ; function 30H: get DOS version
fxn33h:                                 ; function 33H: get/set break flag
fxn36h:                                 ; function 36H: get drive info
fxn3eh:                                 ; function 3EH: close file
fxn42h:                                 ; function 42H: seek
fxn45h:                                 ; function 45H: dup handle
fxn46h:                                 ; function 46H: redirect handle
fxn54h:                                 ; function 54H: get verify flag
fxn57h:                                 ; function 57H: get/set filedate
fxn5ch:                                 ; function 5CH: lock/unlock
fxn68h:                                 ; function 68H: commit file
        pop     bx                      ; restore BX
        call    saveregs                ; unload general registers
```

TINYDOSX.ASM

```
        call    realdos                 ; transfer to DOS
        call    loadregs                ; load general registers & flags
        iret                            ; return to application

                                        ; common handling for functions
                                        ;   passing ASCIIZ addr in DS:DX
fxn39h:                                 ; function 39H: create directory
fxn3ah:                                 ; function 3AH: delete directory
fxn3bh:                                 ; function 3BH: select directory
fxn3ch:                                 ; function 3CH: create file
fxn3dh:                                 ; function 3DH: open
fxn41h:                                 ; function 41H: delete file
fxn43h:                                 ; function 43H: get/set attributes
fxn5ah:                                 ; function 5AH: create temp file
fxn5bh:                                 ; function 5BH: create unique file
        pop     bx                      ; restore BX
        call    saveregs                ; unload general registers
        mov     es,realsel              ; ES:DI = virtual address of
        xor     di,di                   ;         real mode buffer
        mov     si,dx                   ; DS:SI = virtual address of
        cld                             ;         protected mode buffer
@@1:    lodsb                           ; copy ASCIIZ string to
        stosb                           ; real mode buffer
        or      al,al                   ; reached null yet?
        jnz     @@1                     ; no, copy another character
        mov     ax,realseg              ; set address of real mode buffer
        mov     regDS,ax                ; into register data structure
        mov     regDX,0
        call    realdos                 ; transfer to MS-DOS
        call    loadregs                ; load general registers & flags
        mov     dx,protDX               ; restore protected mode DX, ES
        mov     es,protES
        iret                            ; return to application

fxn3fh:                                 ; function 3FH: read file
        pop     bx                      ; restore BX
        call    saveregs                ; unload general registers
        mov     ax,realseg              ; set address of real mode buffer
        mov     regDS,ax                ; into register data structure
        mov     regDX,0
        call    realdos                 ; transfer to MS-DOS
        mov     cx,regCX                ; CX = actual length of data
        push    ds                      ; ES:DI = virtual address of
        pop     es                      ;         protected mode buffer
        mov     di,protDX
        mov     ds,realsel              ; DS:SI = virtual address of
        xor     si,si                   ;         real mode buffer
        cld                             ; copy data from real mode
        rep movsb                       ; buffer to protected mode buffer
        push    es                      ; restore DS = our DGROUP
        pop     ds
        call    loadregs                ; load general registers
        mov     dx,protDX               ; restore protected mode DX, ES
        mov     es,protES
        iret                            ; return to application

fxn40h:                                 ; function 40H: write file
        pop     bx                      ; restore BX
        call    saveregs                ; unload general registers
        mov     es,realsel              ; ES:DI = virtual address of
        xor     di,di                   ;         real mode buffer
        mov     si,dx                   ; DS:SI = virtual address of
        cld                             ;         protected mode buffer
        rep movsb                       ; copy data to real mode buffer
        mov     ax,realseg              ; set address of real mode buffer
        mov     regDS,ax                ; into register data structure
        mov     regDX,0
        call    realdos                 ; transfer to MS-DOS
        call    loadregs                ; load general registers & flags
        mov     dx,protDX               ; restore protected mode DX, ES
        mov     es,protES
        iret                            ; return to application

fxn47h:                                 ; function 47H: get directory
        pop     bx                      ; restore BX
        call    saveregs                ; unload general registers
        mov     ax,realseg              ; set address of real mode buffer
        mov     regDS,ax                ; into register data structure
        mov     regSI,0
        call    realdos                 ; transfer to MS-DOS
        push    ds                      ; ES:DI = virtual address of
        pop     es                      ;         protected mode buffer
        mov     di,protSI
        mov     ds,realsel              ; DS:SI = virtual address of
        xor     si,si                   ;         real mode buffer
        cld
@@2:    lodsb                           ; copy ASCIIZ string from real
        stosb                           ; mode buffer to prot mode buffer
        or      al,al                   ; found null character yet?
        jnz     @@2                     ; no, copy another character
        push    es                      ; restore DS = our DGROUP
        pop     ds
        call    loadregs                ; load general registers
        mov     si,protSI               ; restore protected mode SI, ES
        mov     es,protES
        iret                            ; return to application

fxn00h:                                 ; function 00H: terminate
fxn4ch: pop     bx                      ; function 4CH: terminate
        push    ax                      ; save return code
        mov     ax,0203h                ; restore old GP fault handler
        mov     bl,0dh
        mov     cx,word ptr int0dv+2
        mov     dx,word ptr int0dv
        int     31h
        mov     ax,0205h                ; restore old Int 21H handler
        mov     bl,21h
        mov     cx,word ptr int21v+2
        mov     dx,word ptr int21v
        int     31h
        mov     ax,0101h                ; release real mode buffer
        mov     dx,realsel
        int     31h
        pop     ax                      ; chain to DPMI Int 21H handler
        int     21h                     ; for cleanup and termination

chain:                                  ; general fallthrough point
                                        ; (useful during debugging)
        pop     bx                      ; restore register BX
        jmp     int21v                  ; chain to prev Int 21H owner

doscall endp

;
; Save general registers into real mode data structure for a call to
; real mode routine via DPMI translation services.  Note that segment
; registers are NOT unloaded into structure because they are not valid
; for real mode anyway.
;
saveregs proc   near

        mov     regAX,ax                ; save general registers
        mov     regBX,bx
        mov     regCX,cx
        mov     regDX,dx
        mov     regSI,si
        mov     regDI,di
        mov     regBP,bp
        mov     protDX,dx               ; extra copies for non-
        mov     protSI,si               ; register-based functions
        mov     protES,es
        ret

saveregs endp
;
; Load general registers from real mode data structure.  Note that
; segment registers are NOT loaded because their real mode values
; would cause a GP fault in protected mode.
;
loadregs proc   near

        mov     bp,sp                   ; update CPU flags in
        push    cpuFLAGS                ; stack frame to return
        pop     [bp+6]                  ; DOS function status
        mov     ax,regAX                ; load general registers
        mov     bx,regBX
        mov     cx,regCX
        mov     dx,regDX
        mov     si,regSI
        mov     di,regDI
        mov     bp,regBP
        ret

loadregs endp
;
; Call the DPMI translation function 0300H to simulate a real mode
; software interrupt 21H, transferring control to MS-DOS, passing
; the values stored into the real mode register structure 'regs'.
;
realdos proc    near

        push    es
        mov     ax,0300h                ; DPMI Function 0300H
        mov     bl,21h                  ; software interrupt 21H
        mov     bh,0                    ; flags (bit 0 must be 0)
        mov     cx,0                    ; no. of stack words to copy
        push    ds                      ; ES:DI = address of real
        pop     es                      ; mode register structure
        mov     di,offset DGROUP:regs
        int     31h                     ; to DOS via DPMI host
        pop     es
        ret

realdos endp

  _TEXT ends

        end
```

blow our little programming blunders out of the water with its totally uninformative ''Application has violated system integrity'' dialog box). INITDOSX also allocates a 64K area of conventional memory that the INT 21h handler can later use to pass data back and forth to MS-DOS. Finally, INITDOSX returns control to the C application, which continues its execution in protected mode.

When the C application requests an MS-DOS service, the protected-mode INT 21h handler, named DOSCALL, receives control. DOSCALL saves the flags and registers, then branches through the table DISPATCH to the appropriate subroutine. As you'll notice, I've stubbed out most of the less-common DOS functions to either return an error or abort the application. The functions that TINYDOSX supports, however, are relayed to DOS using the DPMI translation function ''Simulate Real Mode Interrupt.'' Use of this translation function, rather than the speedier DPMI ''raw mode switch'' function, eliminates all sorts of messy problems that are best left to the imagination and experiments of adventurous readers. Naturally, DOSCALL monitors for the fateful INT 21h, function 4Ch, and cleans up after itself accordingly.

Assuming DPMI 0.9 as our platform and linking TINYDOSX directly into the protected-mode application allows us to take some shortcuts that would never suffice in a commercial-grade DOS extender. First, we don't have to build our own loader for the protected-mode application; since TINYDOSX is linked into the application, DOS loads TINYDOSX and the application as a single unit. Second, we don't have to allocate any memory

TINYDOSX.C

```
/*
     TESTDOSX.C --- illustrates use of the DPMI-based
     DOS Extender TINYDOSX to display a message in protected mode

     Copyright (C) 1990 Ziff Davis Communications
     PC Magazine * Ray Duncan

     Build with Microsoft C 6.0 SMALL MODEL as follows:

            MASM TINYDOSX;
            CL TESTDOSX.C TINYDOSX

     Execute under Windows 3.0 in the DOS Box only!
*/

#include <stdio.h>

unsigned extern pascal InitDosX(void);

main()
{
     unsigned saveCS, saveDS;

     _asm mov saveCS,cs          ; store real mode CS
     _asm mov saveDS,ds          ; and DS for display

     printf("\nHello, real mode world! \tCS=%04xh DS=%04xh",
          saveCS, saveDS);

     if(InitDosX())              // attempt mode switch
     {
          puts("\nDPMI initialization failed.");
          exit(1);
     }

     _asm mov saveCS,cs          ; store protected mode CS
     _asm mov saveDS,ds          ; and DS for display

     printf("\nHello, protected mode world! \tCS=%04xh DS=%04xh\n
          saveCS, saveDS);

     _asm mov ah,4ch             ; exit directly to DOS to avoid
     _asm int 21h                ; GP fault in RTL cleanup code
}
```

Figure 2: This is the source code for a simple protected-mode C application that can be linked with TINYDOSX.ASM, listed in Figure 1.

for the application or build any descriptors; we get these services ''free'' when DPMI creates code and data selectors during the initial switch to protected mode. Third, we need to support only those DOS services in our DOS extender that we know our application is actually going to use; we don't have the obligation to translate every known (and unknown) DOS function for protected mode the way a ''real'' DOS extender does.

Suppose we wanted to turn TINYDOSX into a not-so-tiny, more robust DOS extender—where would we start? We'd have to enlarge the support for INT 21h functions to include (at minimum) all the documented MS-DOS services. We'd need to support the immense battery of ROM BIOS services (most of which, luckily, are register-based anyway) and probably, in addition, the Microsoft Mouse INT 33h and the NETBIOS interfaces. We'd want to add more sophisticated facilities for installation of interrupt handlers by the application. And last but not least, we'd be obligated to support most of the different C memory models, a somewhat thorny chore. My first impression of the way to do this would be to incorporate a loader for ''segmented .EXE'' (also called ''new .EXE'') files into our DOS extender, build the application with the Microsoft Segmented Linker, and bind our DOS extender into the .EXE file as the ''real-mode stub.''

TRYING OUT TINYDOSX

TESTDOSX.C, a simple protected-mode C application that can be linked with TINYDOSX.ASM, is shown in Figure 2. To create the executable version of TESTDOSX, enter the following commands:

```
MASM /Mx TINYDOSX;
CL TESTDOSX.C TINYDOSX
```

Be sure that you are using Microsoft 6.0 and compiling for the small model. The resulting application, TESTDOSX.EXE, must be run in one of *Windows* 3.0's DOS boxes so that it has access to DPMI services.

THE IN-BOX

Please send your questions, comments, and suggestions to me at any of the following e-mail addresses:
PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan ■

Computers, FCC Class A, Class B, and You — or When is it better to get a B than an A?

You need to know the difference between computers that meet the FCC class B radio frequency emissions standards and those that meet only the Class A standards.

Computers emit radio signals in their operation. Because these signals may cause interference to radio and television reception, the marketing and the use of computers is regulated by the Federal Communications Commission. Under federal rules, computer users are responsible for remedying interference, including interference in neighboring homes.

Computers certified by the FCC as meeting the Class B standard are less likely to cause interference to radio and TV reception than those that have been verified by the manufacturer or importer to the Class A standards. Only Class B certified computers may be advertised, sold, or leased for use in residences. A similar regulatory program applies in Canada.

Buyers seeking computers for use in homes (including offices at home) should shop for computers and peripherals which have been Class B certified. These devices carry a label with an FCC ID number. Both new and used Class A verified devices may be sold only for use in commercial and industrial locations. Signals from computers are more likely to be masked by electrical noise from other equipment in such an environment. These areas are also likely to have fewer radios and TVs. Accordingly, equipment marketed only for use in these locations may meet the less rigorous Class A standard. Class B certified equipment may be marketed for use in residences as well as commercial and industrial locations.

As you shop for a computer for use in your home, look for the FCC classification in the specifications or ask your vendor to recommend only machines that have been certified to the Class B limits. TV viewers and radio listeners in your home and in neighboring homes will be glad you did.